Mandelbrot Mini-project

Group MATTEK6-4
Daniel Bernard van Diepen
& Dennis Grøndahl Andersen,

April 2020

1 Introduction

The Mandelbrot set is a quadradic complex mapping described by

$$z_{i+1} = z_i^2 + c, \quad i = 0, 1, \ldots, I-1 \tag{1}$$

where $c \in \mathbb{C}$ and $z_i \in \mathbb{C}$ for $i = 0, 1, \ldots, I-1$, i.e I iterations of eq. (1). The initial condition z_0 is

$$z_0 = 0 + 0j \tag{2}$$

For each point c we compute I iterations of (1) and determine

$$\iota(c) = \min \mathbb{T}, \quad \mathbb{T} = \{i || z_i| > T, i = 1, \ldots, I\} \cup \{I\} \tag{3}$$

where T is a chosen tolerance. The function $\iota(c)$ yields the minimum iteration of (1) for which $|z_i| > T$ for a given complex point c. The initial condition is always chosen such that $|z_0| < T$, so the lower bound of $\iota(c)$, i.e minimum amount of iterations, is 1. Furthermore the upper bound of $\iota(c)$ is I, since if $|z_{i+1}| \leq T$ for all i we would have $\mathbb{T} = \{\emptyset\} \cup \{I\} = \{I\}$. For plotting purposes we define the linear map

$$\mathcal{M}(c) = \frac{\iota(c)}{I}. \tag{4}$$

This mapping yields some information of stability of the point c in the form of how fast $|z_{i+1}|$ increases for each iteration i. If the value $\mathcal{M}(c)$ is close to 1, then that would imply that $|z_i| < T$ for close to all i, i.e the point is stable, while if the value of $\mathcal{M}(c)$ is very small it would indicate that $|z_i| > T$ happens at a small index i, i.e the point is unstable. A point c is said to be part of the Mandelbrot set if $|z_{n+1}|$ is bounded for $n \to \infty$.

The objective of this mini-project is to determine $\mathcal{M}(c)$ for a c-mesh. This mesh is a subset of the complex plane such that $c \in \mathbb{C}^{[-2,1]\times[-1.5,1.5]}$, ie $-2 \leq \Re(c) \leq 1$ and $-1.5 \leq \Im(c) \leq 1.5$. we choose p_{re} and p_{im} points from this subset for $\Re(c)$ and $\Im(c)$, respectively. The subset of the complex plane is then described by the following complex matrix $\mathbf{C}$

$$C = \begin{bmatrix} -2.0 & \ldots & 1.0 \\ \vdots & & \vdots \\ -2.0 & \ldots & 1.0 \end{bmatrix} + j \cdot \begin{bmatrix} 1.5 & \ldots & 1.5 \\ \vdots & & \vdots \\ -1.5 & \ldots & -1.5 \end{bmatrix} \in \mathbb{C}^{p_{re} \times p_{im}} \tag{5}$$

2 Software design

Style convention

Variables: For global constants we use the uppercase snake case convention, e.g `SNAKE_CASE`, while for local variables we use lowercase, e.g `snake_case`.

Functions: For functions we use lowercase snake case.

Functionality

Function description on I/O and core functionality level. Based on the theory, we can split the functionality into three functions.

Input:

Data type	Name	Description
list	`RE_INTERVAL`	Real interval
list	`IM_INERVAL`	Imaginary interval
int	`ITER_MAX`	The maximum amount of iterations in the function $\iota(c)$
float	`TOLERANCE`	The tolerance T which $\|z_i\|$ needs to stay lower than
int	`P_RE`	The dimension of $\mathcal{M}$ in the real (horizontal) direction
int	`P_IM`	The dimension of $\mathcal{M}$ in the imaginary (vertical) direction

Table 1: Inputs to the program

Output:

Data type	Name	Description
image	`m_colormesh_(method).pdf`	Colormesh plot of $\mathcal{M}$
float	`TIME_EXEC`	Execution time of the script
hdf	`C_MESH`	The C-mesh
hdf	`m_mesh_matrix`	Matrix of the linear map for each c

Table 2: Outputs of the program

Test: The Mandelbrot set is known and constant. Therefore we can test the output of the script by visually comparing the colormesh plot with known Mandelbrot sets like the example shown in the miniproject specification.

3 Naive Implementation

Documentation

We will first describe the functions on I/O and core functionality level. Based on the theory, we can split the functionality into three functions.

Functions:

- `c_mesh(re_Interval, im_interval, p_re, p_im)`
  - Generates mesh of `p_im` $\times$ `p_re` evenly spaced complex points in the subset of the complex plane.
  - Input: Intervals `re_interval`, `im_interval`, and `p_re`, `p_im`
  - Output: nd array `C_MESH`
- `iota(complex_point, tolerance, iter_max)`
  - Calculates the complex mapping described in (1) and then the function $\iota(c)$ as described in (3)
  - Input: complex number `complex_point`, float `tolerance`,
  - Output: integer iteration number i for which $|z_{i+1}| > T$ for point c
- `m_map(iter_stop,iter_max)`
  - Calculates the linear map $\mathcal{M}(c)$ as described by (4)
  - Input: Integer `iter_stop` the iteration for which $|z_{i+1}| > T$, integer `Iter_max` the maximum amount of iterations.
  - Output: float `iter_stop`/`iter_max`

Algorithm

From the theory, we can devise an algorithm

Algorithm 1 Naive Implementation

```
   Input: RE_INTERVVAL, IM_INTERVAL, ITER_MAX, TOLERANCE, P_RE, P_IM
1: C_MESH = c_mesh(RE_INTERVAL, IM_INTERVAL, P_RE, P_IM)
2: for n in {0, ..., P_IM} do
3:     for m in {0, ..., P_RE} do
4:         iterations[n,m] = iota(C_MESH[n,m], TOLERANCE,
                                  ITER_MAX)
5:         m_mesh_matrix[n,m] = m_map(iterations[n,m])
6: plot(M)
```

Hence in the naive implementation we generate the nd array c-mesh with a call of the `c_mesh()` function. Then we sequentially, i.e for each complex point of the c-mesh, call the `iota()` function to get the index for which $|z_{i+1}| > T$ and then use this index in the call of the `M_map()` function to perform the linear map in (4). The result is then saved as an entry of the matrix `m_mesh_matrix`, which is a matrix in $\mathbb{R}^{p_{re} \times p_{im}}$. This matrix is then used as an input to the `matplotlib` library color mesh function `pcolormesh()` with the specified restricted subset of the complex plane.

Output

The output of the script is as mentioned the colormap made from the matrix `m_mesh_matrix`. The output is from a tolerance of $T = 2$, $I = 100$ iterations and a resolution of $p_{re} = 1000$ and $p_{im} = 1000$. The color map is illustrated in figure 1.

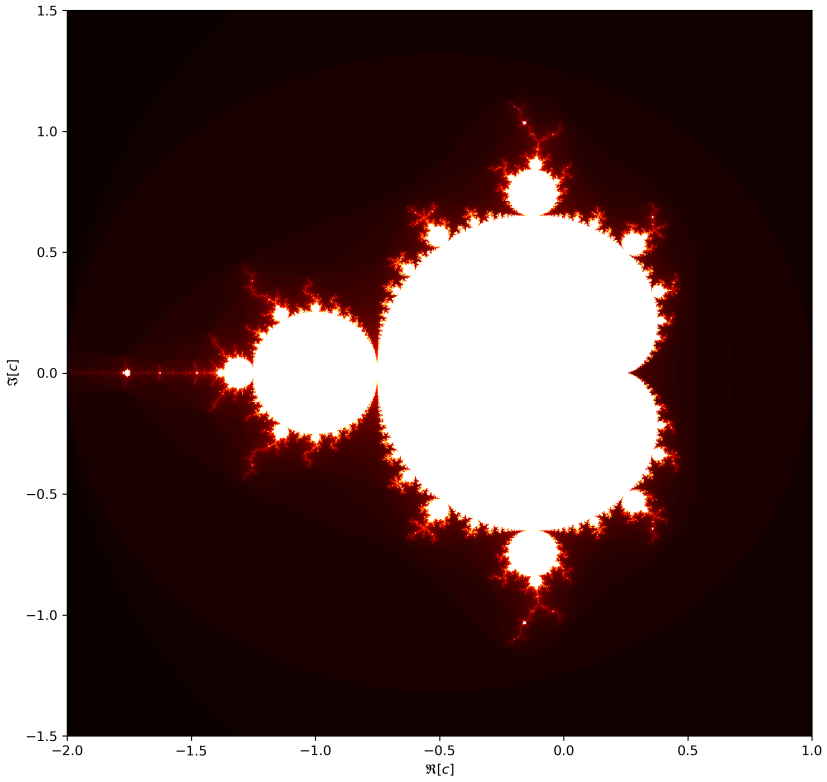


Figure 1: Mandelbrot set created with the naive implementation

As seen this looks very close to the example Mandelbrot set in shown in the miniprojekt proposal.

Line Profiler

In order to evaluate code performance, we use the line profiler built into Spyder. In the following figure the result of the line profiler analysis for running the code with the given subset of the complex plane, a tolerance of $T = 2$, $I = 100$ maximum iterations and with a resolution of $p_{re} = 1000$ and $p_{im} = 1000$ corresponding to $1,000,000$ points.

Function/Module	Total Time	Local Time	Calls
iota	25.50 sec	23.53 sec	1000000
savefig	2.29 sec	25.80 us	1
_find_and_load	1.18 sec	7.26 ms	789
m_map	128.46 ms	128.46 ms	1000000
pcolormesh	117.66 ms	13.50 us	1
figure	67.59 ms	36.10 us	1
testmod	63.59 ms	31.30 us	1
c_mesh	42.79 ms	42.67 ms	1
__init__	9.51 ms	38.00 us	1
create_dataset	5.27 ms	136.40 us	5
linspace	4.23 ms	92.00 us	65
__exit__	638.20 us	7.80 us	1
xlabel	148.00 us	6.50 us	1
<built-in method numpy.zeros>	131.40 us	131.40 us	9
<built-in method time.time>	48.60 us	48.60 us	208
ylabel	43.90 us	3.20 us	1
<built-in method builtins.print>	11.30 us	11.30 us	1
get_cmap	10.90 us	7.40 us	4
__enter__	600.00 ns	600.00 ns	1

Figure 2: Line profiler result for the naive implementation

Not surprisingly almost all of the time of the execution is spent in the function `iota`, since this is where (1) is calculated for every point c. Furthermore, for each iteration we have a conditional verification if $|z_{i+1}| > T$, which is also time consuming.

4 Vectorized implementation

In this implementation we utilize vectorized computations to optimize the naive implementation. As seen in the line profiler the biggest improvement we can make is in the iota function. The main idea for optimising with vectorization is that instead of computing (1) for each point iteratively as we do in the naive implementation we compute the complex mapping of the entire c-mesh at once in a vectorised manner, yielding a nd array with dimension $p_{re} \times p_{im} \times I$ and thereafter check for which iterations i where $|z_{i+1}| > T$ manifesting as an nd array with the same dimension. This is then flattened to the minimum iteration where $|z_{i+1}| > T$. Furthermore, we implemented some vectorisation in the creation of the c-mesh and instead of spending time on calling the `m_map()` function we instead implemented this directly into the vecorised iota function.

Documentation

As with the naive implementation we first describe the functions on an I/O and core functionality level.

Functions:

- `c_mesh(re_interval, im_interval, p_re, p_im)`
  - Generates mesh of `p_im` $\times$ `p_re` evenly spaced complex points in the subset of the complex plane.
  - Input: Intervals `re_interval`, `im_interval`, and `p_re`, `p_im`
  - Output: nd array `C_MESH`
- `iota_vector(c_mesh, tolerance, iter_max,p_re,p_im)`
  - Calculates the complex mapping described in (1) for the entire c-mesh. This is done by calculating all z_i, $i \in [0, \texttt{iter\_max})$ generating an nd array. This nd array is flattened to its minimum values over the iterations.
  - Input: nd array `c_mesh`, float `tolerance`, integer `iter_max`, integer `p_re`, integer `p_im`
  - Output: The mandelbrot set values for the given complex points, normalized for the maximum amount of iterations.

Algorithm

Algorithm 2 Vectorised Implementation

Input: `RE_INTERVAL`, `IM_INTERVAL`, `ITER_MAX`, `TOLERANCE`, `P_RE`, `P_IM`

```
1: C_MESH = c_mesh(RE_INTERVAL, IM_INTERVAL, P_RE, P_IM)
2: M_MESH = iota_vector(C_MESH, TOLERANCE, ITER_MAX, P_RE,
                        P_IM)
3: plot(M_MESH)
```

Output

Similarly to the naive implementation the output is the color mesh. The output for tolerance $T = 2$, $I = 100$ maximum iterations and a resolution $p_{re} = 1000$, $p_{im} = 1000$ is illustrated in figure 3.

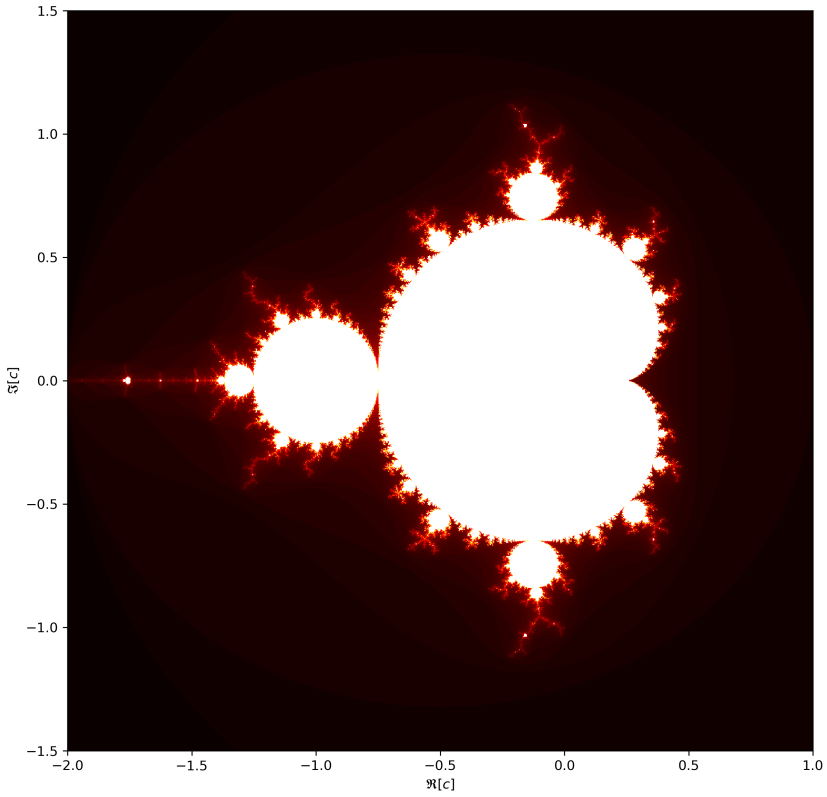
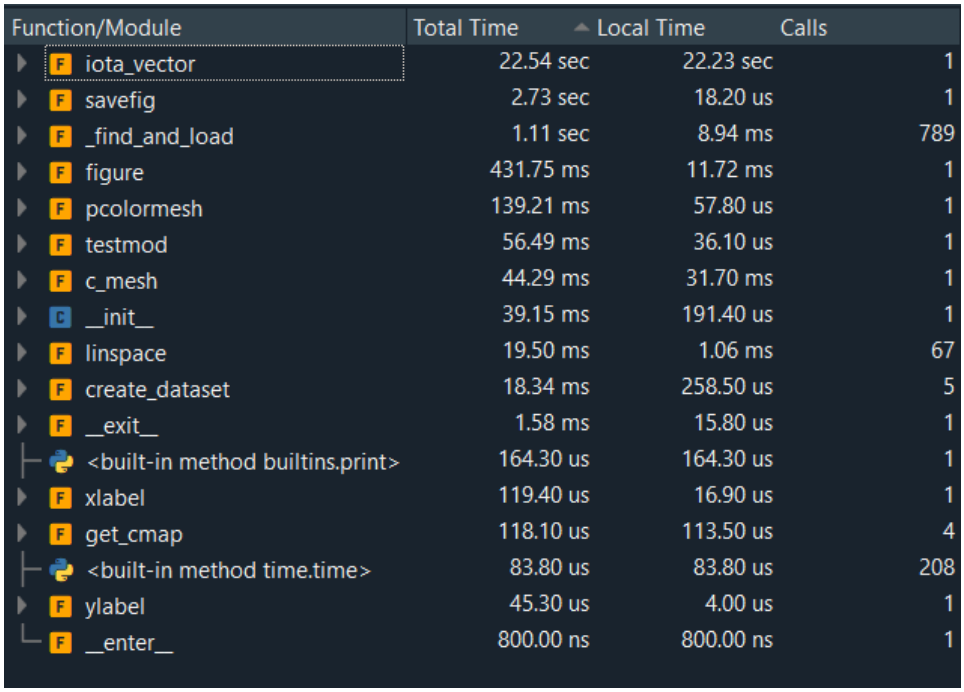


Figure 3: Mandelbrot set created with the vectorised implementation

Line profiler

The line profiler result for the vectorised implementation is shown in the following figure



Function/Module	Total Time	Local Time	Calls
iota_vector	22.54 sec	22.23 sec	1
savefig	2.73 sec	18.20 us	1
_find_and_load	1.11 sec	8.94 ms	789
figure	431.75 ms	11.72 ms	1
pcolormesh	139.21 ms	57.80 us	1
testmod	56.49 ms	36.10 us	1
c_mesh	44.29 ms	31.70 ms	1
__init__	39.15 ms	191.40 us	1
linspace	19.50 ms	1.06 ms	67
create_dataset	18.34 ms	258.50 us	5
__exit__	1.58 ms	15.80 us	1
<built-in method builtins.print>	164.30 us	164.30 us	1
xlabel	119.40 us	16.90 us	1
get_cmap	118.10 us	113.50 us	4
<built-in method time.time>	83.80 us	83.80 us	208
ylabel	45.30 us	4.00 us	1
__enter__	800.00 ns	800.00 ns	1

Figure 4: Line profiler result for the vectorised implementation

Like the naive implementation most of the execution time is spent in the vectorised `iota()` function, `iota_vector()`. However, compared to the naive implementation it only spends 14.58 seconds instead of 23.37 seconds in the naive implementation. Hence, some optimisation has been achieved.

5 Multiprocessing implementation

For the multiprocessing implementation we first had to decide what to implement multiprocessing on. As it would seem from the line profiler results from both the naive and vectorised implementation the majority of execution time is spend computing (1). This computation can be implemented in a multiprocessing manner by sending a subset of points to workers, which then computes z_i of the given points independently for $i = 1, \ldots, I$ and send

the results back for evaluation of which index $|z_{i+1}| > T$ if any. We have implemented this in two ways. One for a fixed $n = 6$ number of cores with respect to the capability of our hardware, and an implementation which allowed for difference in number of cores $n = 1, \ldots, 6$, such that we can see the difference each core makes for the total execution time.

Documentation

Function description on I/O and core functionality level. Focus on difference from the previous two.

Functions:

- `c_mesh(re_interval, im_interval, p_re, p_im)`
  - Generates mesh of `p_im` $\times$ `p_re` evenly spaced complex points in the subset of the complex plane.
  - Input: Intervals `re_interval`, `im_interval`, and `p_re`, `p_im`
  - Output: nd array `C_MESH`
- `iota(complex_point, tolerance, iter_max)`
  - Calculates the complex mapping described in (1) and then the function $\iota(c)$ as described in (3)
  - Input: complex number `complex_point`, float `tolerance`, integer `iter_max`, the maximum number of iterations.
  - Output: integer iteration number i for which $|z_{i+1}| > T$ for point c
- `iota_vec(tolerance,iter_max, c_vec)`
  - Calculates the function ι for a vector and applies the linear map $\mathcal{M}(c)$ as described by (4)
  - Input: float `tolerance`, integer `iter_max`, the maximum number of iterations, and `c_vec` consisting of one row of the C matrix.
  - Output: float 1-d numpy array constituting a row of the mandelbrot mesh matrix.

Algorithm

Algorithm 3 Multiprocessing Implementation

Input: `RE_INTERVAL`, `IM_INTERVAL`, `ITER_MAX`, `TOLERANCE`, `P_RE`, `P_IM`

1: `CPU_COUNT` = no. of cpus
2: `C_MESH = c_mesh(RE_INTERVAL, IM_INTERVAL, P_RE, P_IM)`
3: Create `pool` of `CPU_COUNT` workers
4: Apply SIMD to `pool`. Instruction: `iota_vec`. Data: rows of `C_MESH`
5: `M_MESH` = result of `pool`
6: `plot(M_MESH)`

Output

As with the other implementations we show the output in the form of the color mesh, with the same values for T, I, p_{re} and p_{im}.

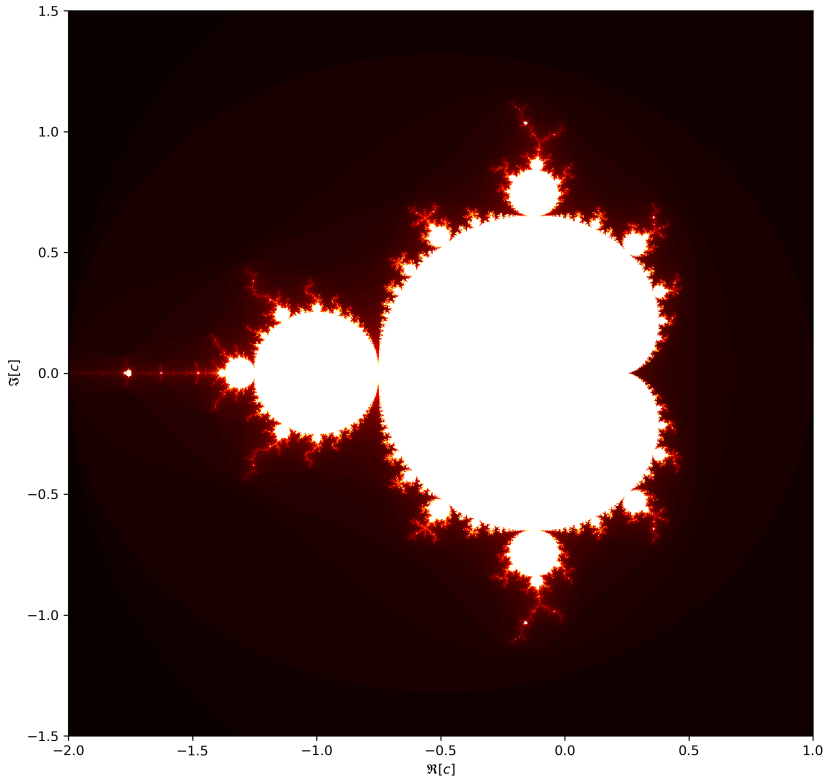


Figure 5: Mandelbrot set created with the multiprocessing implementation.

Amount of cores

In the following figure we have plotted the execution time versus the amount of cores used in the multiprocessing implementation to see the computational improvement

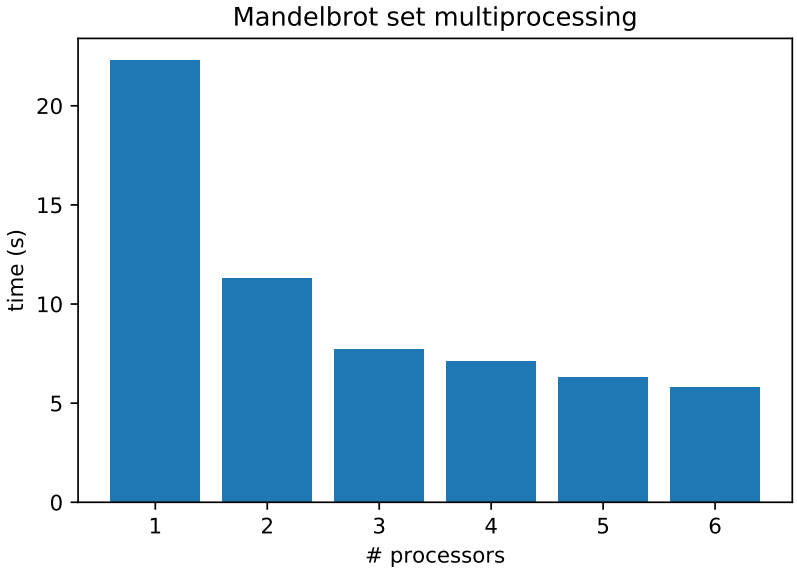


Figure 6: Execution time, versus amount of cores in the multiprocessing implementation. Averaged over 10 realizations. CPU used: AMD 6350 6-core.

It is seen that a significant improvement in execution time happens when the amount of cores is increased from 1 to 2. However, it is seen that beyond this point, the difference is negligible.

6 Comparison of implementations

We value the quality of the implementation by the execution time. We therefore run each implementation 10 times with a tolerance of $T = 2$, $I = 100$ and a resolution of $p_{re} = 1000$ and $p_{im} = 1000$. Furthermore the amount of cores in the multiprocessing implementation is set to 6. For each run we get the execution time, and lastly we average the result. The averaged results can be seen in the following table

Method	time (s)
Naive	21.219
Vectorized	13.217
Multiprocessing	5.7875

Table 3: Averaged execution time over 10 executions for the different methods. CPU used: AMD 6350 6-core.

As seen the fastest implementation is the multiprocessing implementation, followed by the vectorised implementation and lastly the naive implementation. Furthermore in the following bar graph the significance of improvement can be seen from implementation.

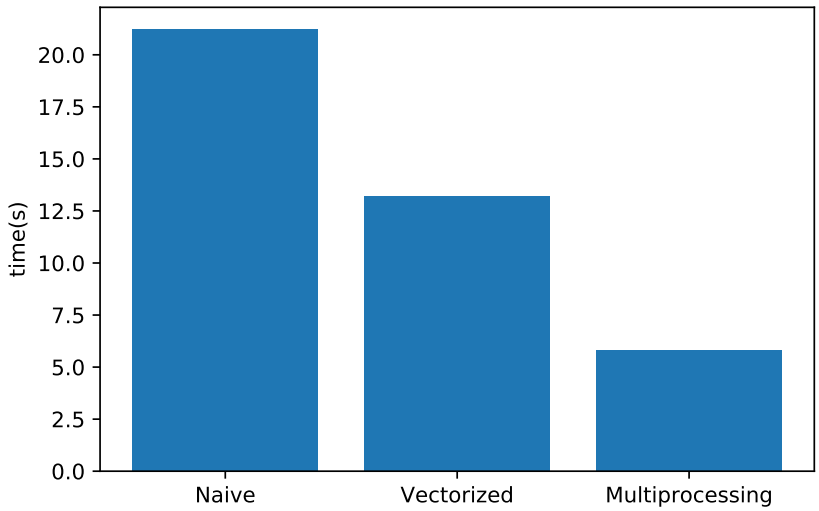


Figure 7: Comparison of execution time. CPU used: AMD 6350 6-core.

7 Evaluation

In conclusion the least efficient implementation was the naive, which took an average execution time of 21.219 seconds. The vectorised implementation improved upon this execution time with an average execution time of 13.217 seconds. Lastly, the multipocessing implementation with 6 cores took an average of 5.7875 seconds, improving on both the naive and vectorised implementation. Furthermore, it was seen in figure 6 that there were diminishing returns to increasing the number of cores, as increasing the amount of cores beyond 3 yielded only a negligible improvement of execution time as compared to the increase from 1 cores to 2.